CE143: COMPUTER CONCEPTS & PROGRAMMING

Chapter – 09

User-Defined Functions

Devang Patel Institute of Advance Technology and Research

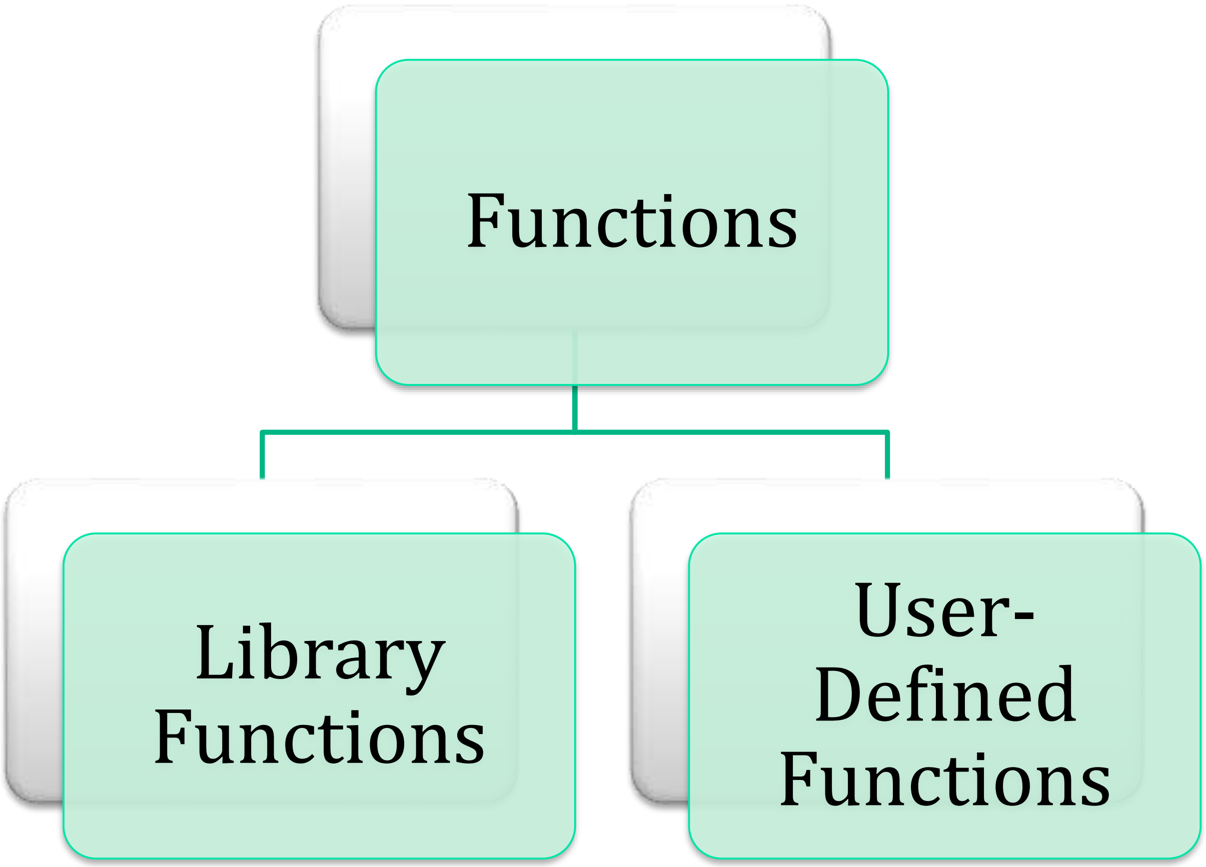
Objectives

- To be able to outline user-defined functions
- To be able to identify the elements of user-defined functions
- To be able to explain the different categories of functions
- To be able to know the concept of recursion
- To be able to describe how arrays are passed to functions
- To be able to discuss the relevance of storage classes on scope, visibility and lifetime of variable.

Introduction

- In this chapter, we will discuss
  - How a function is designed?
  - How a function is integrated into a program?
  - How two or more functions are put together?
  - How they communicate with one another?

Introduction



Ex: printf, scanf

Ex: main

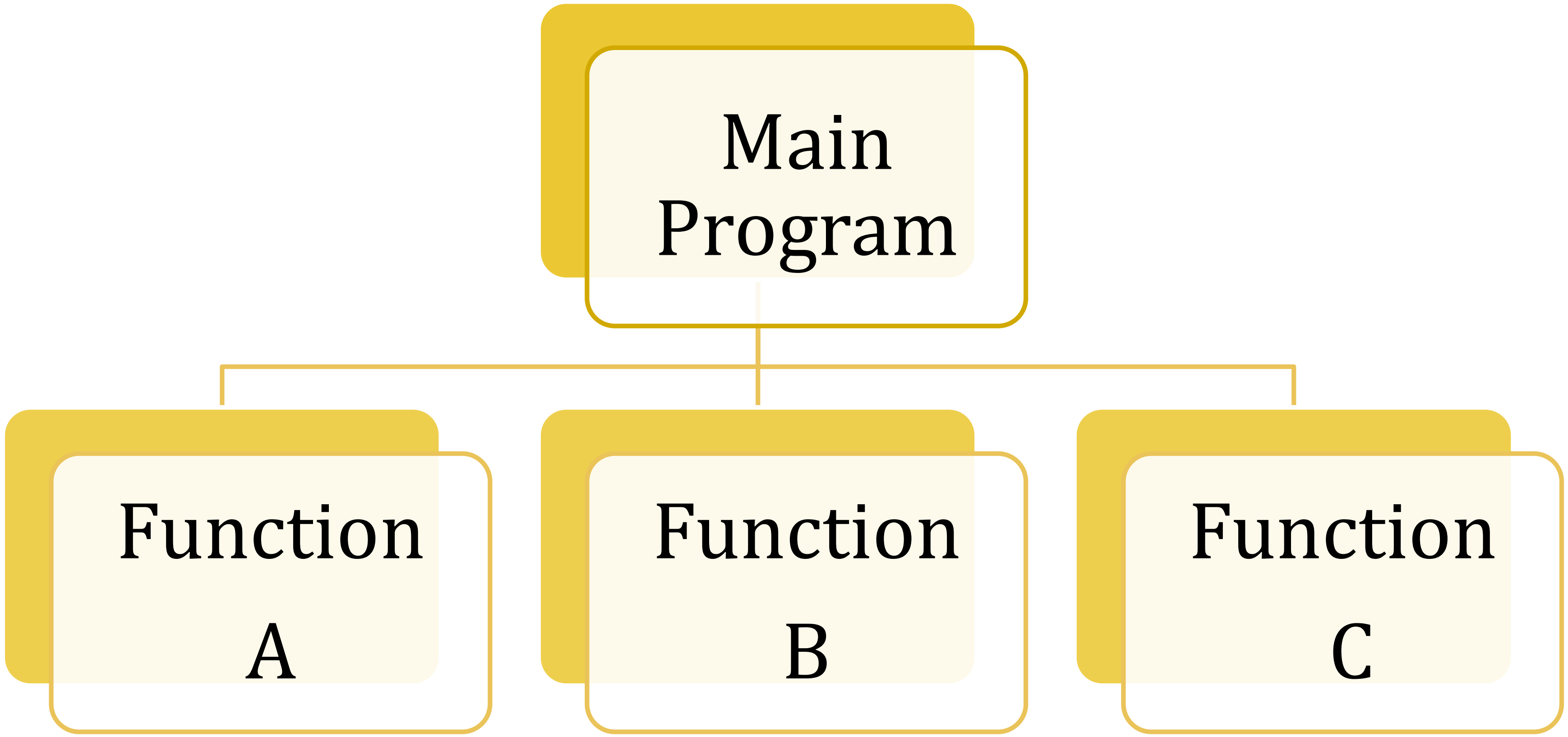
Need for User-Defined Functions

- main is specially recognized function, which is used to indicate start of program execution.
- By using only main following problems may occur:
  - Programs becomes too large
  - Higher complexity
  - Debugging, testing and maintaining becomes difficult
- If program is divided into functional parts, then each part can be independently coded and combined into single unit

Need for User-Defined Functions

- Independently coded programs are called **"Subprograms"**.
- They are easy to understand, debug and test.
- Subprograms are referred as **"functions"** in C.
- There are times when certain operations and calculations are repeated in program.
  - For Example: Factorial of a number
- For such requirements a function can be designed and used whenever required.

Top-down Modular programming



Modular programming Characteristics

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that don’t have a calling-called relationship.
5. All modules are designed as single-entry, single-exit systems using control structures.

Advantages of Modularization

- High level logic of the overall problem is solved first.
- Length of a source program can be reduced.
- Easy to locate and isolate a faulty function.
- Function my be used by many other programs.
- Instead of starting all over again from scratch, programmers can use already built functions.

Elements of user-defined functions

1. Function Definition
2. Function Call
3. Function Declaration

Definition of functions

- A function definition also known as function implementation shall have the following elements:
  1. Function Name
  2. Function Type (Return Type)
  3. List of Parameters
  4. Local Variable Declarations
  5. Function Statements
  6. A Return Statement
- All these elements are grouped together into two sections:
  1. Function Header (First three elements)
  2. Function Body (Second three elements)

Definition of functions

```
returnType  functionName  (type1  argument1, type2  argument2, ...)
{
        //Local Variable Decalarations
        //Body of the function
        //Return statement
}
```

Formal Parameter List

- The *formal parameters list* declares the variables that will receive the data sent by the calling function

- The *function type* or the *return type* specifies the data type of the value that the function is expected to return to the calling function.

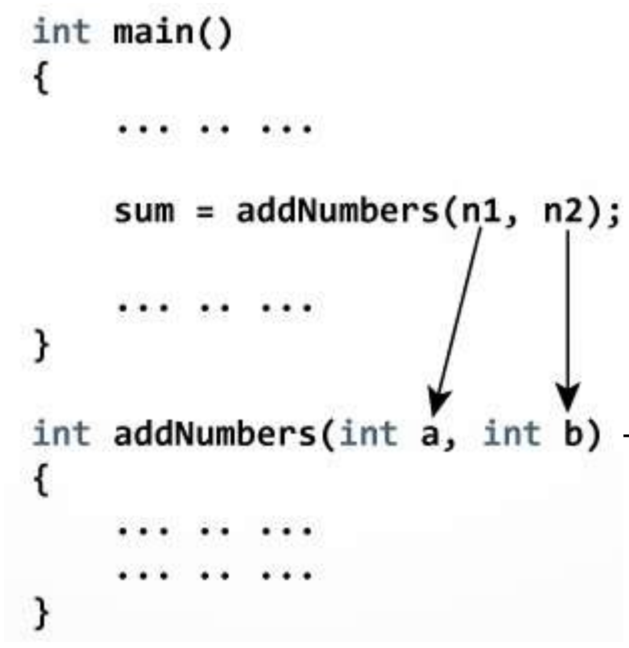
Return Statement

- A return statement immediately transfers control of the program to the calling function.
- It can be placed anywhere in the function body.
- It can have the following forms:
  - return;
    - simple, returns no data, acts like a closing brace
  - return(expression);
    - returns the value of the expression to the calling function
- A return statement can also be used in the following way:
  - If(condition)

    return;

Function Call

- A function can be called by simply writing the name of the function followed by a list of *actual parameters* (arguments) in parenthesis and then terminating the statement with a semicolon.
- The *actual arguments* are passed to the *formal arguments* in the function definition and used there.

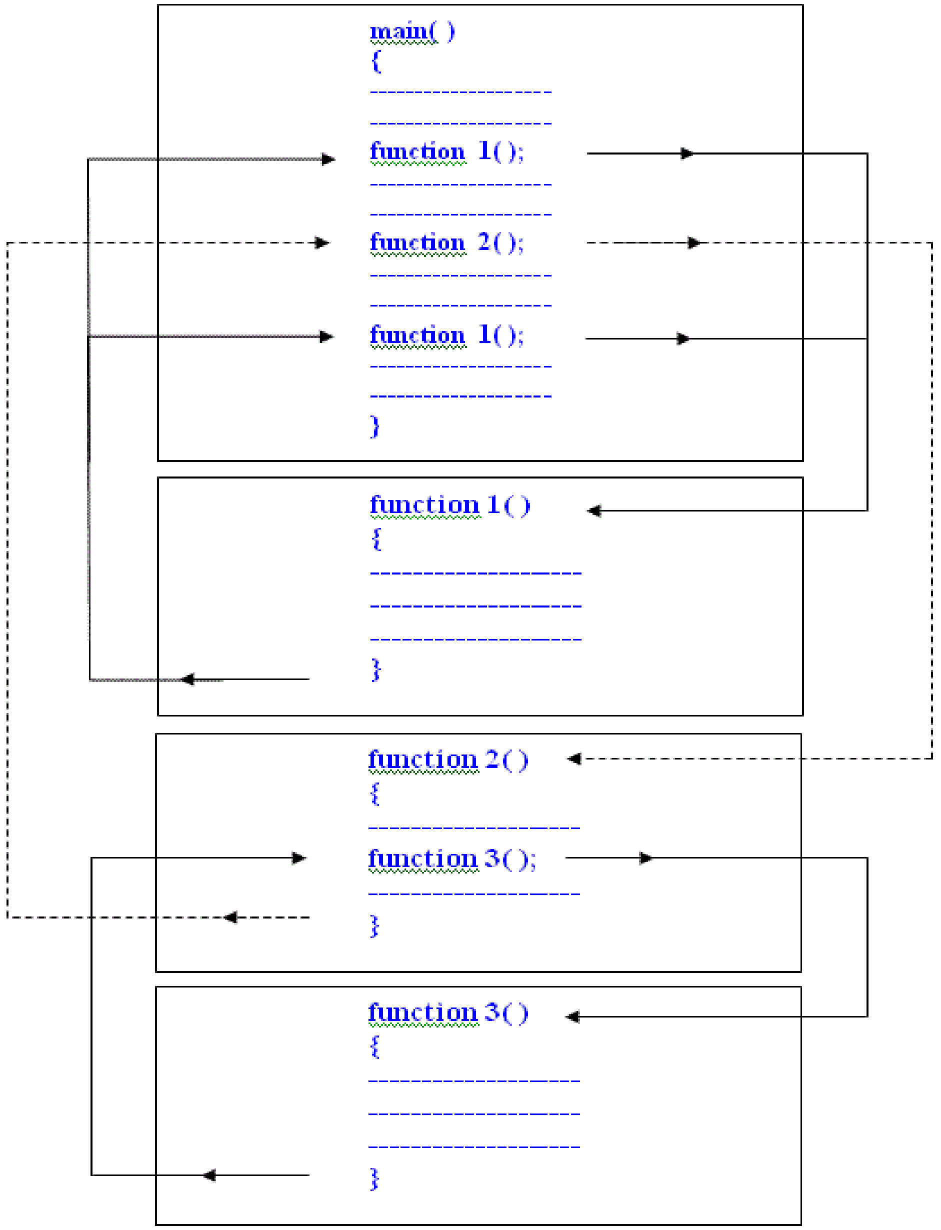


```
int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

n1, n2 are actual arguments

a, b are formal arguments

Flow of control in a multi-function Program



Contains the function call

Calling Function

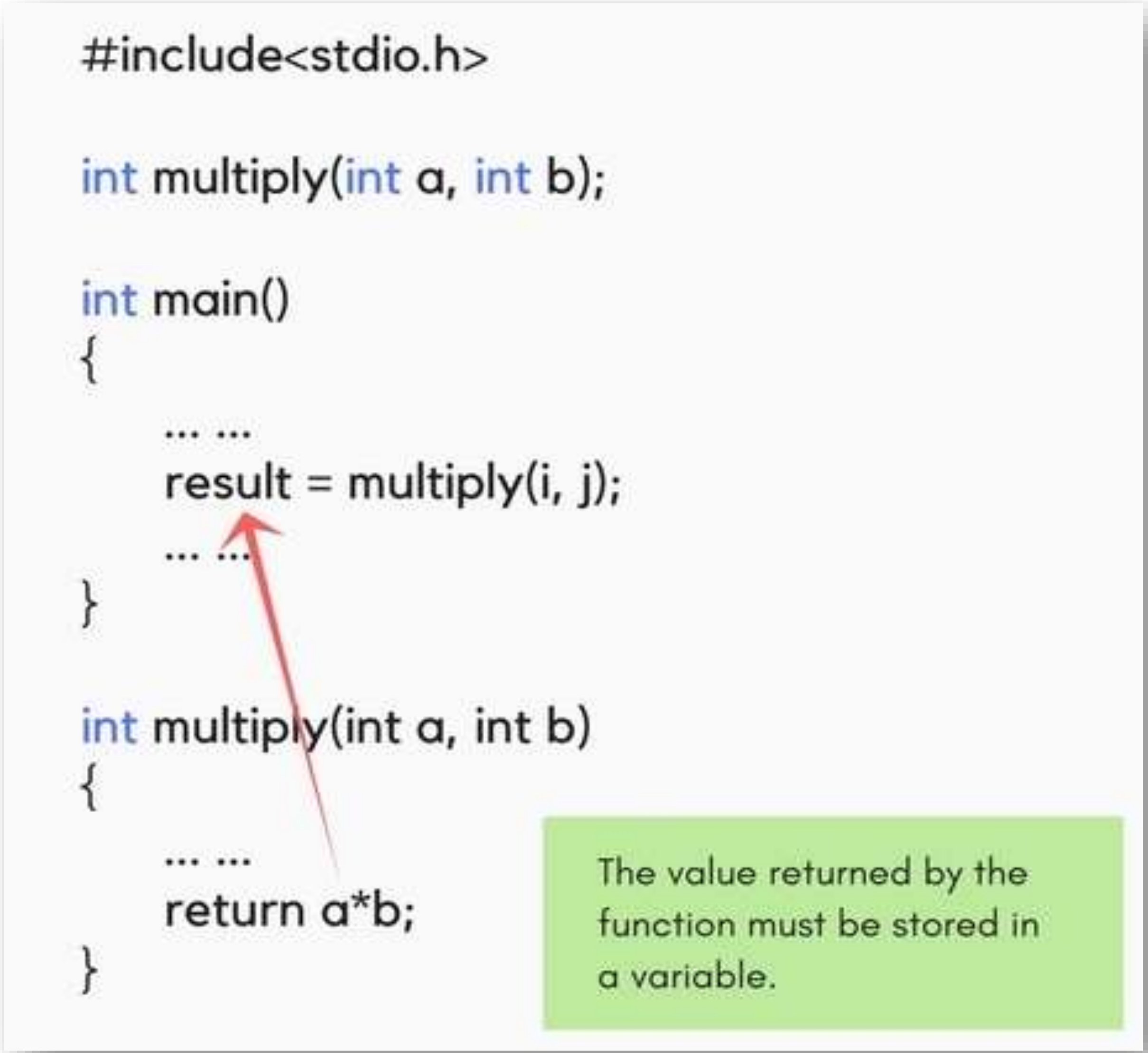
The name of the function in the call

Called Function

Function Declaration/ Prototype

- Equally acceptable prototypes:
  - int mul(int , int);
  - int mul(int a, int b);
  - mul(int, int);

Return Values in Functions



```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
    return a*b;
}
```

The value returned by the function must be stored in a variable.

Categories of Functions

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and one return value.
4. Functions with no arguments but return a value.
5. Functions with multiple return values.

Now, from next slide onwards we will see four different programming examples doing the same thing but using different categories of functions.

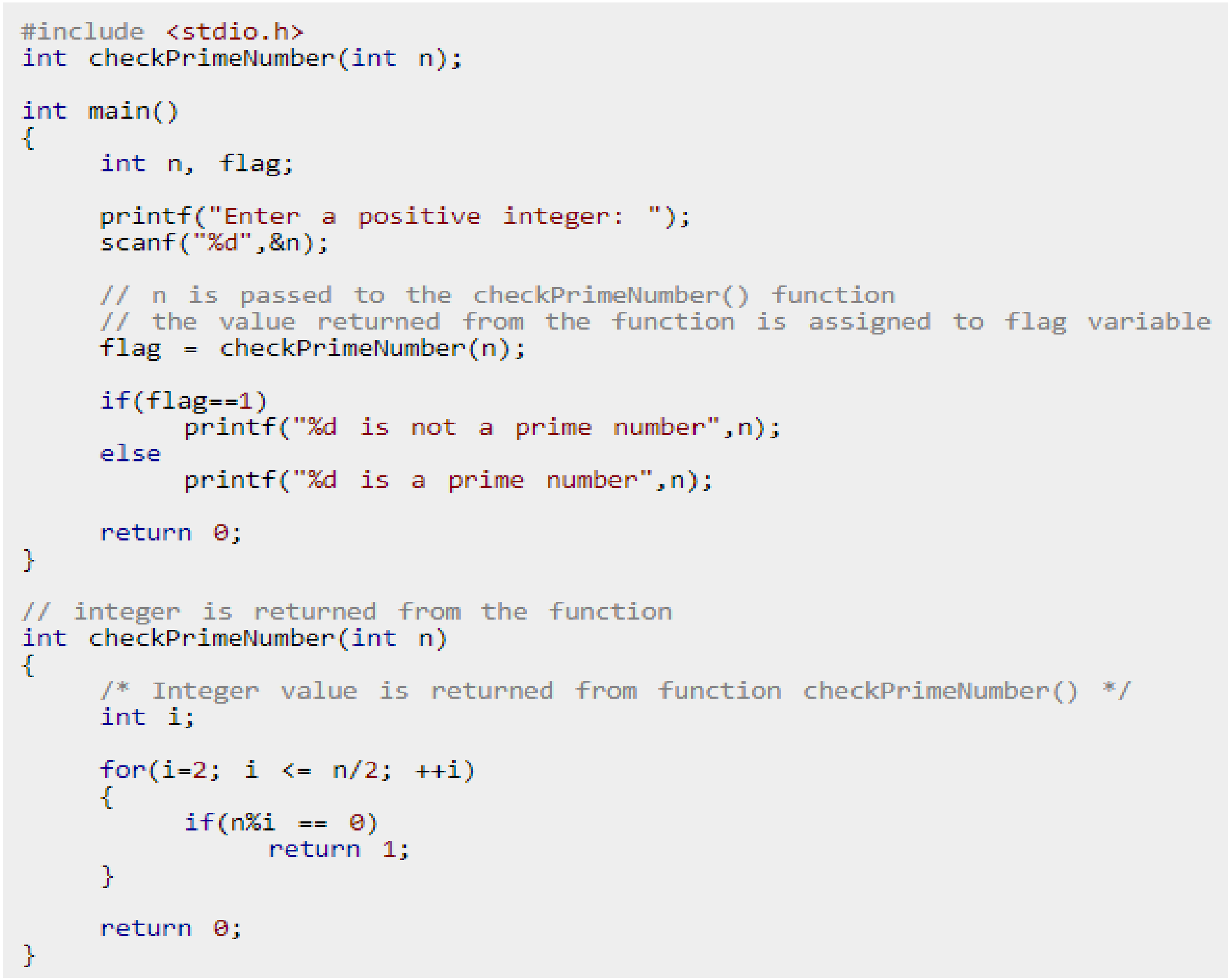
Functions with no arguments and no return values

```
#include <stdio.h>

void checkPrimeNumber();

int main()
{
     checkPrimeNumber();     // no argument is passed to prime()
     return 0;
}

// return type of the function is void becuase no value is returned
void checkPrimeNumber()
{
     int n, i, flag=0;

     printf("Enter a positive integer: ");
     scanf("%d",&n);

     for(i=2; i <= n/2; ++i)
     {
          if(n%i == 0)
          {
               flag = 1;
          }
     }
     if (flag == 1)
          printf("%d is not a prime number.", n);
     else
          printf("%d is a prime number.", n);
}
```

Functions with arguments and no return values

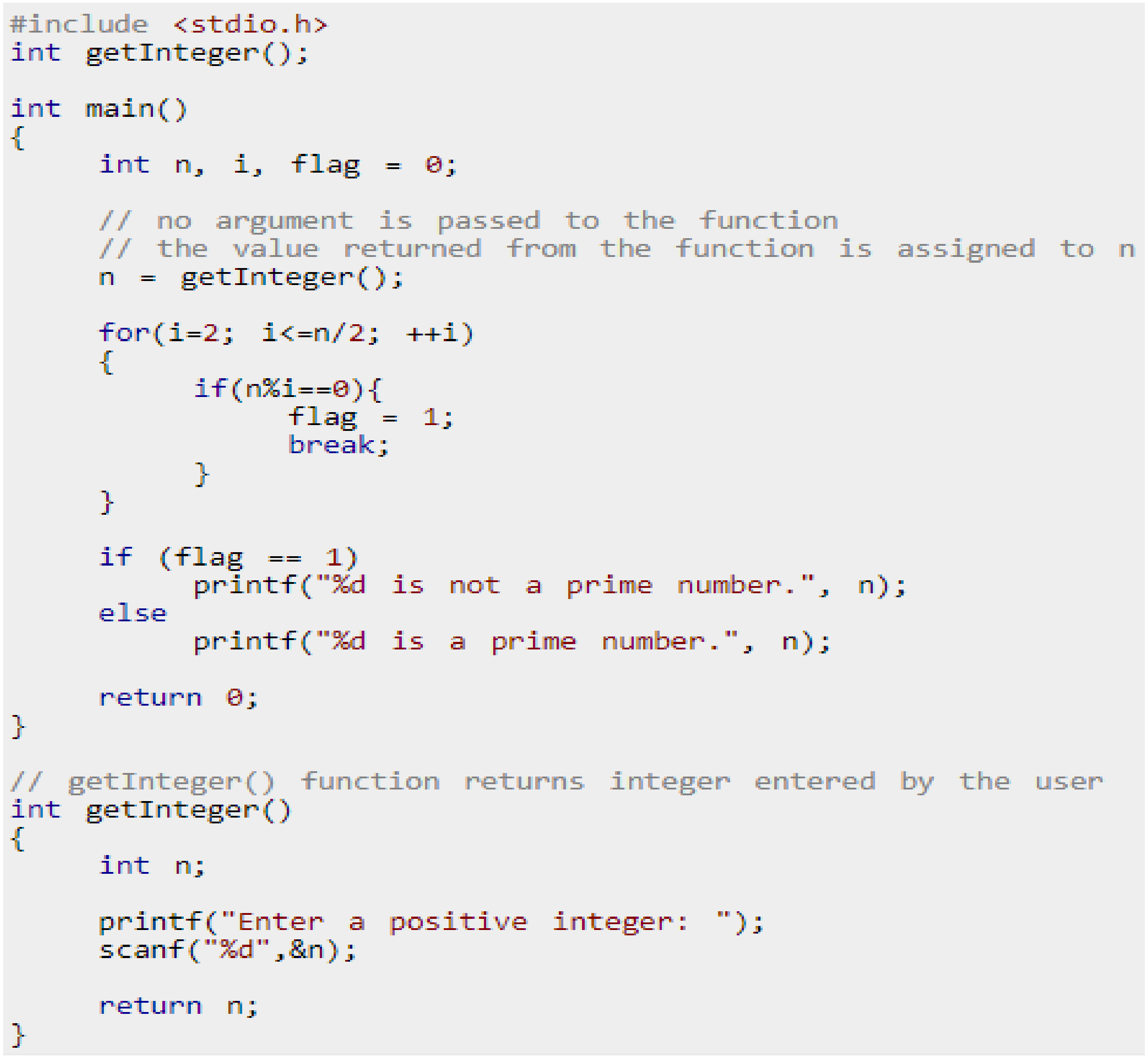
```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main()
{
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;
}

// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n)
{
    int i, flag = 0;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```

Functions with arguments and one return value



```c
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
      int n, flag;

      printf("Enter a positive integer: ");
      scanf("%d",&n);

      // n is passed to the checkPrimeNumber() function
      // the value returned from the function is assigned to flag variable
      flag = checkPrimeNumber(n);

      if(flag==1)
            printf("%d is not a prime number",n);
      else
            printf("%d is a prime number",n);

      return 0;
}

// integer is returned from the function
int checkPrimeNumber(int n)
{
      /* Integer value is returned from function checkPrimeNumber() */
      int i;

      for(i=2; i <= n/2; ++i)
      {
            if(n%i == 0)
                  return 1;
      }

      return 0;
}
```

Functions with no arguments but returns a value



```c
#include <stdio.h>
int getInteger();

int main()
{
	int n, i, flag = 0;

	// no argument is passed to the function
	// the value returned from the function is assigned to n
	n = getInteger();

	for(i=2; i<=n/2; ++i)
	{
		if(n%i==0){
			flag = 1;
			break;
		}
	}

	if (flag == 1)
		printf("%d is not a prime number.", n);
	else
		printf("%d is a prime number.", n);

	return 0;
}

// getInteger() function returns integer entered by the user
int getInteger()
{
	int n;

	printf("Enter a positive integer: ");
	scanf("%d",&n);

	return n;
}
```

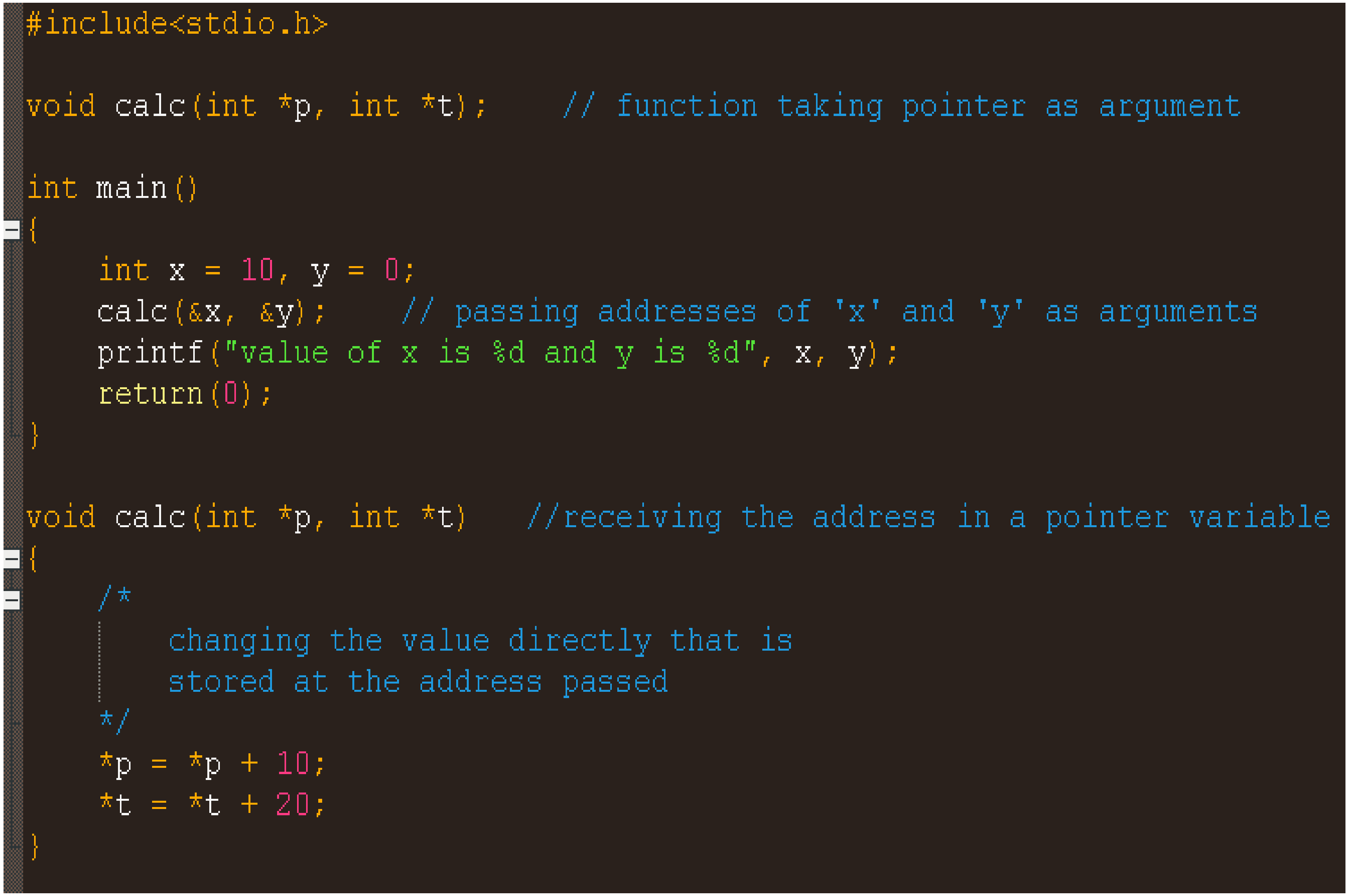
Which approach is better?

- It depends on the problem you are trying to solve.
- In case of this problem, the last approach is better.
- A function should perform a specific task. The checkPrimeNumber() function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not, which makes code modular, easy to understand and debug.

Passing data to Functions

- Pass by Values (also known as Call by Value)
  - In call by values, only the copy of the values of variables is passed to the called function and that function works with the copies of the values and not on the original variables.
- Pass by Address (also known as Call by Pointers, sometimes mistakenly called Call by Reference)
  - In call by pointers, the addresses of the variables are passed to the called function and hence any changes made to the values using pointer variables will change the original values of the passed variables.

Functions returning multiple values (call by pointers/ reference)



```c
#include<stdio.h>

void calc(int *p, int *t);    // function taking pointer as argument

int main()
{
    int x = 10, y = 0;
    calc(&x, &y);    // passing addresses of 'x' and 'y' as arguments
    printf("value of x is %d and y is %d", x, y);
    return(0);
}

void calc(int *p, int *t)   //receiving the address in a pointer variable
{
    /*
        changing the value directly that is
        stored at the address passed
    */
    *p = *p + 10;
    *t = *t + 20;
}
```

Nesting of Functions

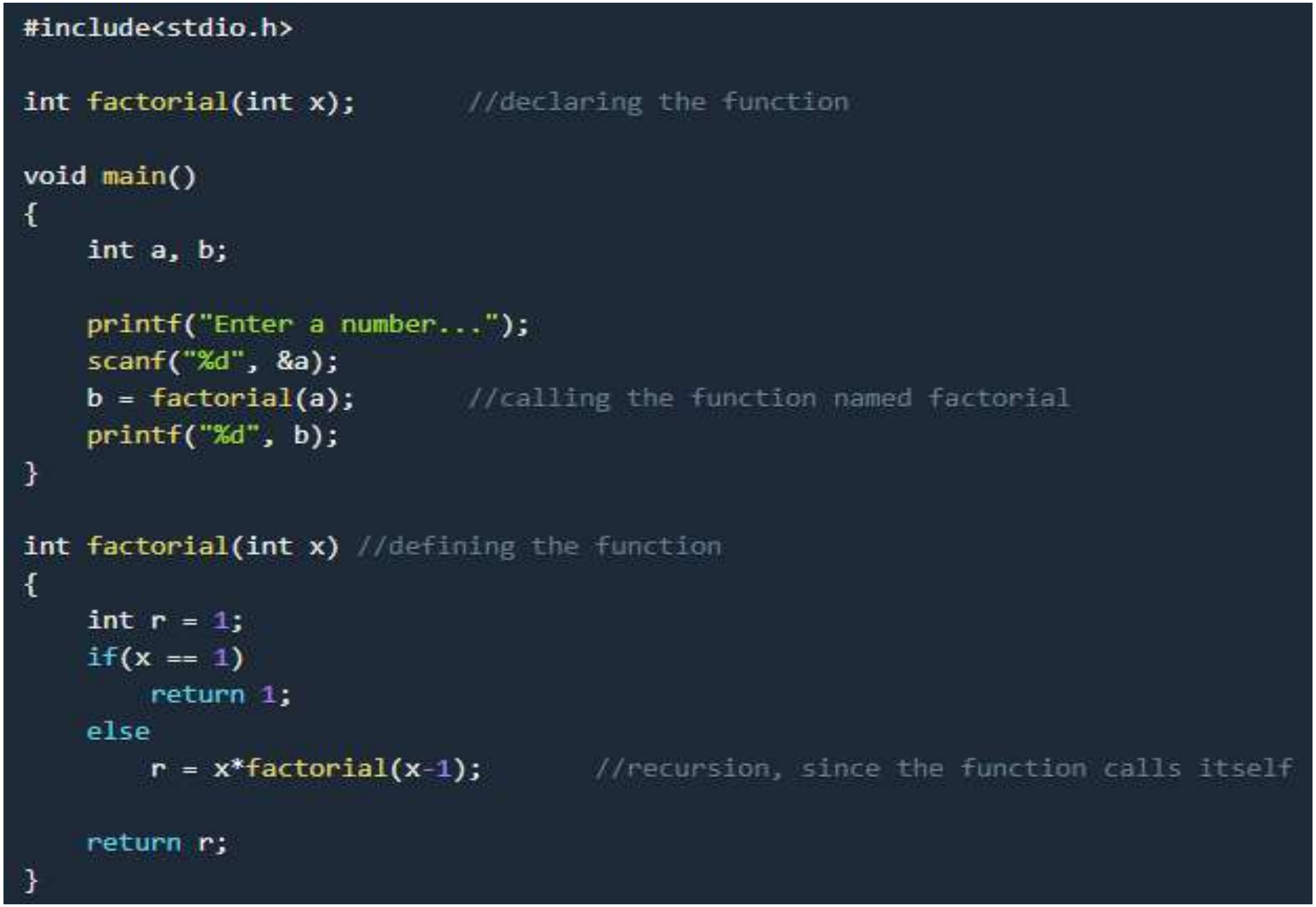
```
function1()
{
        // function1 body here
        function2();
        // function1 body here
}
```

- Nesting of functions is calling one function from inside the body of another.
- Careful when nesting functions as this can lead to infinite nesting.

Recursion

- Recursion is a special case of nested functions in which a function calls itself from inside it’s own body.
- We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.
- In the following slide we will see an example of how to use recursion to find the factorial of a number.

Recursion



```
#include<stdio.h>

int factorial(int x);       //declaring the function

void main()
{
    int a, b;

    printf("Enter a number...");
    scanf("%d", &a);
    b = factorial(a);       //calling the function named factorial
    printf("%d", b);
}

int factorial(int x) //defining the function
{
    int r = 1;
    if(x == 1)
        return 1;
    else
        r = x*factorial(x-1);       //recursion, since the function calls itself

    return r;
}
```

Recursion Pros and Cons

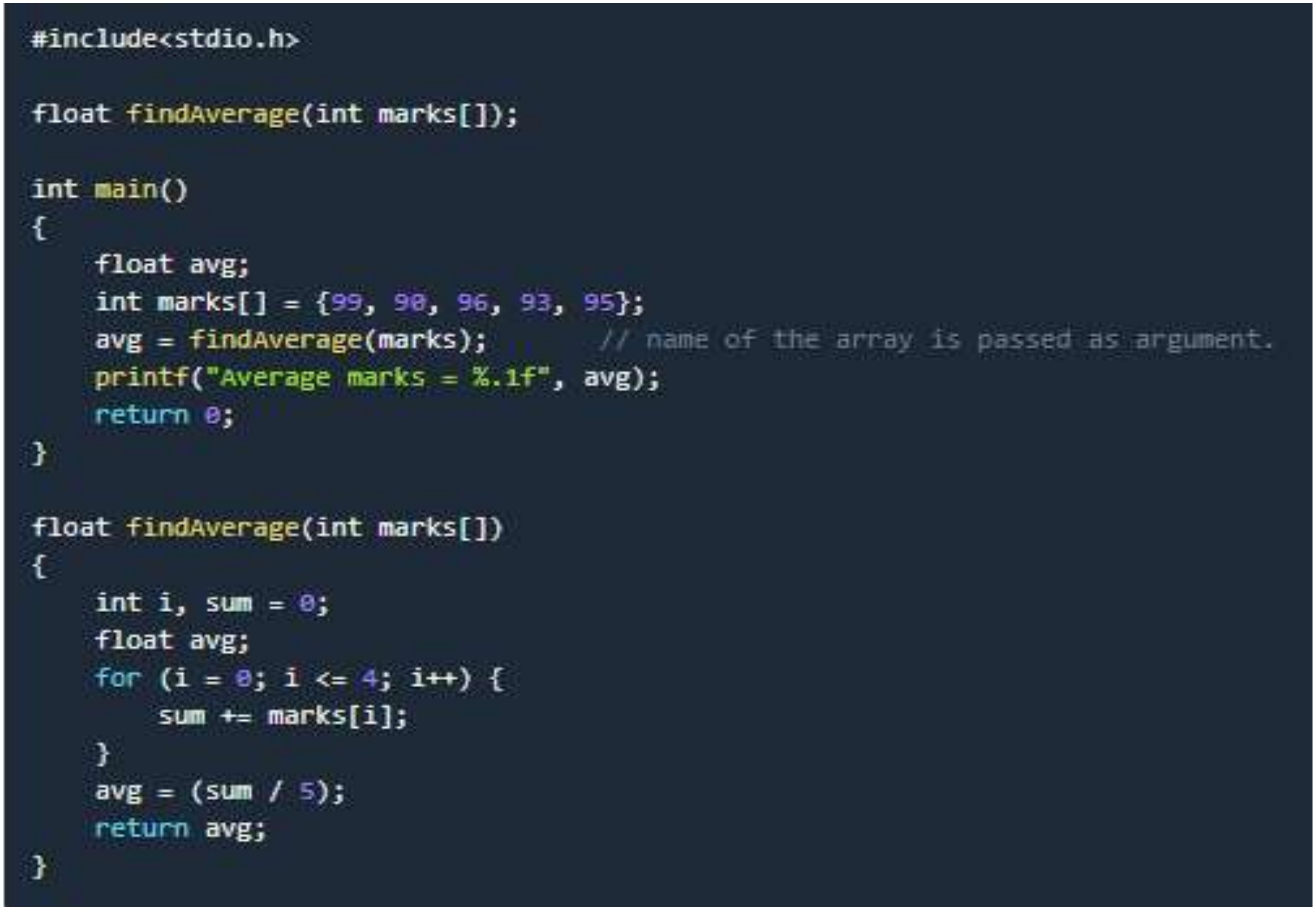
- Recursive functions can be effectively used to solve problems where solution can be expressed in terms of successively applying that same solution to subsets of the problem.

- Recursion can reduce time complexity. But may increase it if not used properly.
- Recursion can reduce the size of the code and make debugging easier.
- Recursion uses more memory during execution of the program.

Passing arrays to Functions as Arguments

- We will only send in the name of the array as argument, which is nothing but the *address of the starting element of the array*, or we can say the starting memory address.
- In the function definition and the prototype, the formal parameter must be of array type.
- The size of the array does not need to be specified in the function definition or prototype.
- Same rules also apply for 2D arrays to be passed as arguments except that the function definition/ prototype must specify the size of at least the first dimension of the array that is to be passed.

Passing arrays to Functions as Arguments



```c
#include<stdio.h>

float findAverage(int marks[]);

int main()
{
    float avg;
    int marks[] = {99, 90, 96, 93, 95};
    avg = findAverage(marks);       // name of the array is passed as argument.
    printf("Average marks = %.1f", avg);
    return 0;
}

float findAverage(int marks[])
{
    int i, sum = 0;
    float avg;
    for (i = 0; i <= 4; i++) {
        sum += marks[i];
    }
    avg = (sum / 5);
    return avg;
}
```

Previous Year Questions

- Write a program to create a user-defined function prime () which would find whether the entered number is prime or not.
- Explain how to pass an Array to a function as an argument with example.
- Every C program must have at least one user defined function. **True or False?** Answer: True, Main function is user defined.
- By default, __________ is the return type of user defined function.
- A variable declared inside a function by default assumes __________ storage class.

- What is function prototype? Is it optional?
- Explain ‘function with argument no return type’ in detail with example.
- A recursive function is always infinite without terminating condition. True or False?
- Explain the advantages of using user-defined function in C.
- What is recursive function? Explain with suitable example.
- Explain Call by value and Call by address with example.